

Compiler Design Project 1

Semester 6 | Jan 20 2018

Nihala Karapoola | 15CO231 || Pushpita Patil | 15CO235

# Introduction to a Compiler

A compiler is a program which processes statements written in a particular programming language and turns them into the code that a computer's processor uses (machine language). The original code file that the user creates contains what are called the source statements. The programmer then runs the appropriate language compiler, specifying the name of the file that contains the source statements.

While executing, the compiler first analyzes all of the language statements syntactically in one or more successive stages, builds the output code, making sure that statements that refer to other statements are referred to correctly in the final code. Traditionally, the output of the compilation has been called object code or sometimes an object module. The object code is machine code that the processor can execute.

The difference between a compiler and an assembler is that a compiler works with what are sometimes called 3GL and higher-level languages. An assembler works on programs written using a processor's assembler language.

The process in Compiling goes as follows:

# Phase 1: Lexical Analysis

The aim of lexical analysis or scanning is to read the stream of characters making up the source program from left to right and group them into tokens. Tokens are sequences of characters with a collective meaning. There are usually only a small number of tokens for a programming language: constants (integer, double, char, string, etc.), operators (arithmetic, relational, logical), punctuation, and reserved words.

Specific instances of tokens are called lexemes. Or in other words, a lexeme is the actual character sequence forming a token, while the token is the general class that a lexeme belongs to. Some tokens have exactly one lexeme, like ">"; for others, there are many lexemes, like integer constants.

The scanner has to determine how the input stream can be divided into valid symbols in the source language, but doesn't know which token should come where. Very few errors can be detected at the lexical level alone because the scanner only has a localized view of the source program and has no context.

The scanner can report about characters that are not valid tokens (like an illegal or unrecognized symbol) and a few other malformed entities (like illegal characters within a string constant, unterminated comments). It does not look for or detect garbled sequences, tokens out of place, undeclared identifiers, misspelled keywords, mismatched types and the like.

# Our Mini- Compiler

Our lexical analyzer understands and creates a symbol table for

* Data types: int, float, char, enum
* User defined data types: struct
* Constants: int, float, string
* Operators: arithmetic and relational
* Conditional statements: if, else, switch
* Control statements: continue, break;
* Loops: for, while
* File inclusions
* Comment: single line, multi line
* Function: prototype, declaration, call
* Bracket nesting
* Delimiter ;

# Test cases

## Error Test Case 1

//Invalid identifier error

#inlcude <stdio.h>

void main()

{

int a;

char z?1

flot b=12.2;

if(a=1)

{

b=13.3;

}

else

c=12

for(a=1;a<2;a++)

{

b=b/2;

}

switch ( b )

{

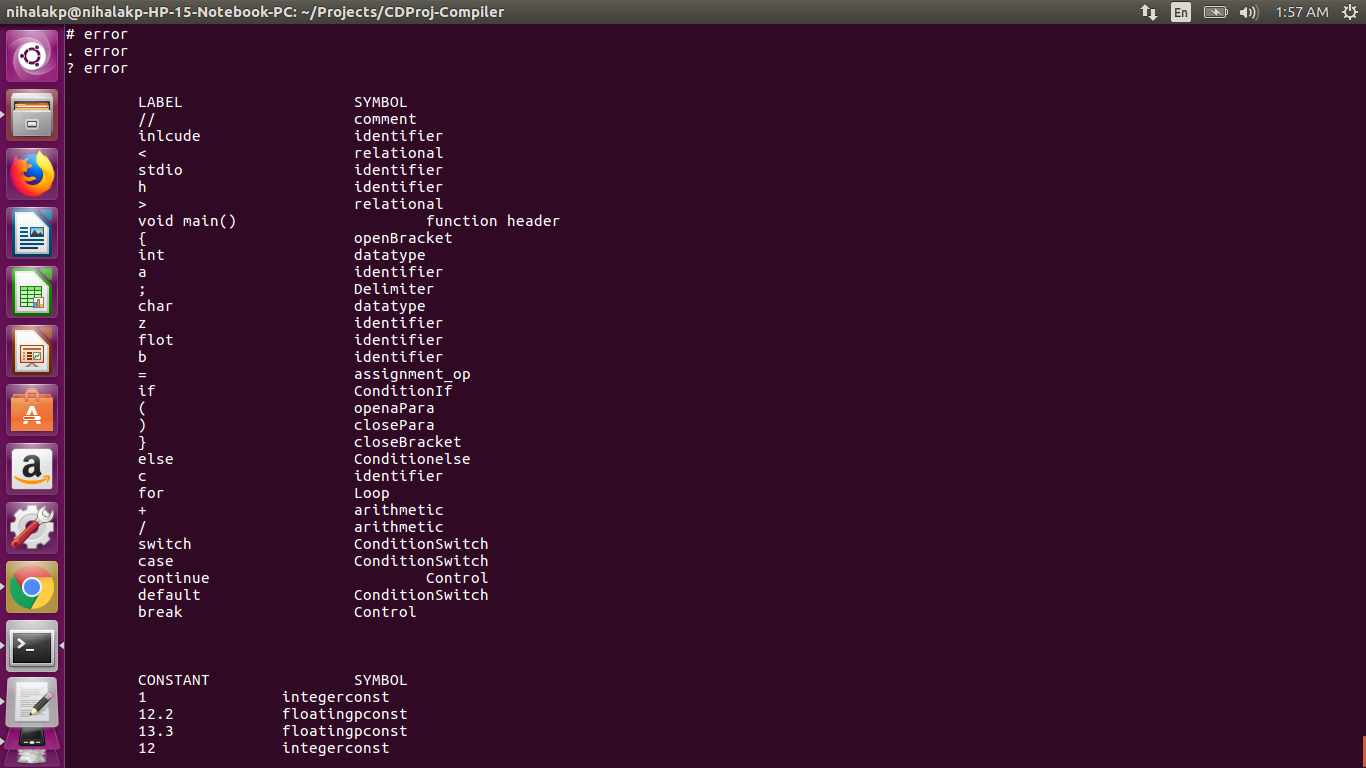
case 1: a = 1 ;

case 2: continue ;

default: break;

}

}



* The misspelled include statement is noted as identifier, and the file that was meant to be included is treated as relation operators and identifiers
* The misspelled float datatype is noted as identifier
* The rest of the tokens are properly stored, and the constant table is made properly

## Error Test Case 2

//Invalid function header error

# include <stdio.h>

voiddd testcase()

{

int x;

}

void main()

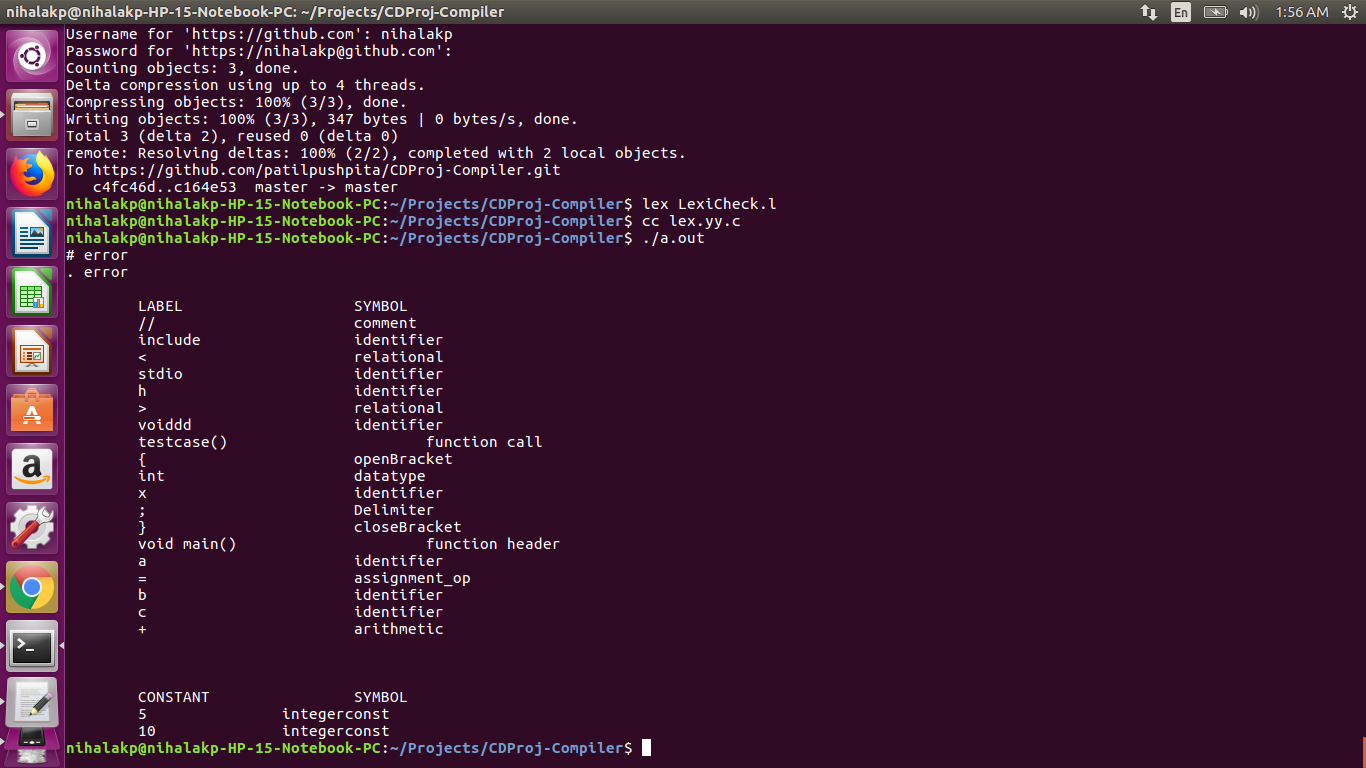
{

int a = 5;

int b=10;

int c = a+b;

}



* The include file line is misspelled and is treated as a combination of identifiers and relational operators
* One improper function definition is identified as an identifier and function call and the proper function definition is properly identified

## Error Test Case 3

//Parenthesis Error

#

#include<stdio.h>

void main()

{

int x,y,z;

int i;

x=1;

y=5;

z=7;

for(i=0 ; i<n;++i)

{

printf("Test");

}

}

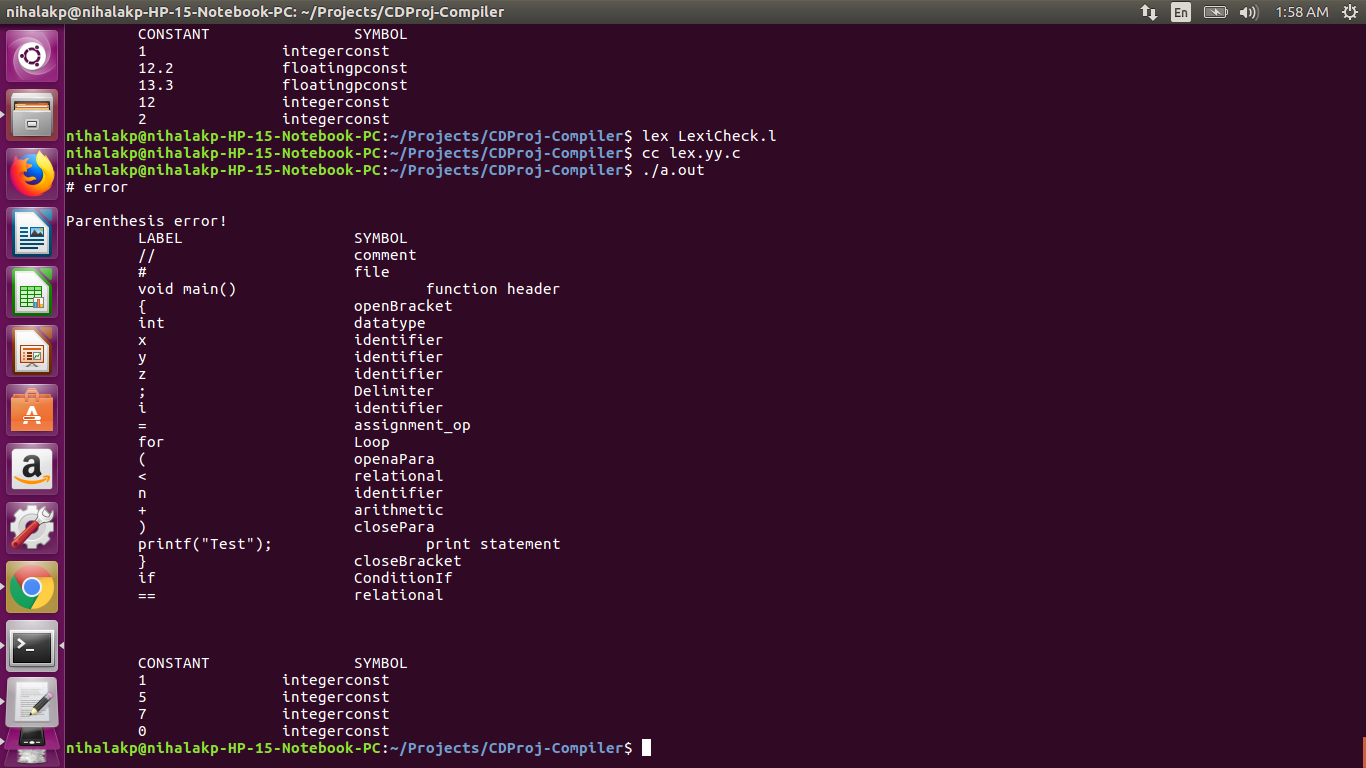
if(x == y)

{

{printf("Test");}

}

}



* When the parentheses are mismatched, the error is displayed.
* Printf is identified
* All other tokens are identified properly

## No Error Test Case

//No error file

/\* The code includes every acceptable case\*/

#include <stdio.h>

void example1();

void example2()

{

int a;

a = 3;

}

void example1()

{

printf("Example1");

example2();

}

/\*multiline comment

nihala

pushpita\*/

void main()

{

int a , b ;

float c ;

char d ;

a = 10 ;

c = 10.12 ; //comment1

if ( a = 10 )

{

b = 1 ;

}

else

{

c = 11.12 ;

}

//comment2

while ( b > 0 )

{

b = b - 1 ;

}

switch ( b )

{

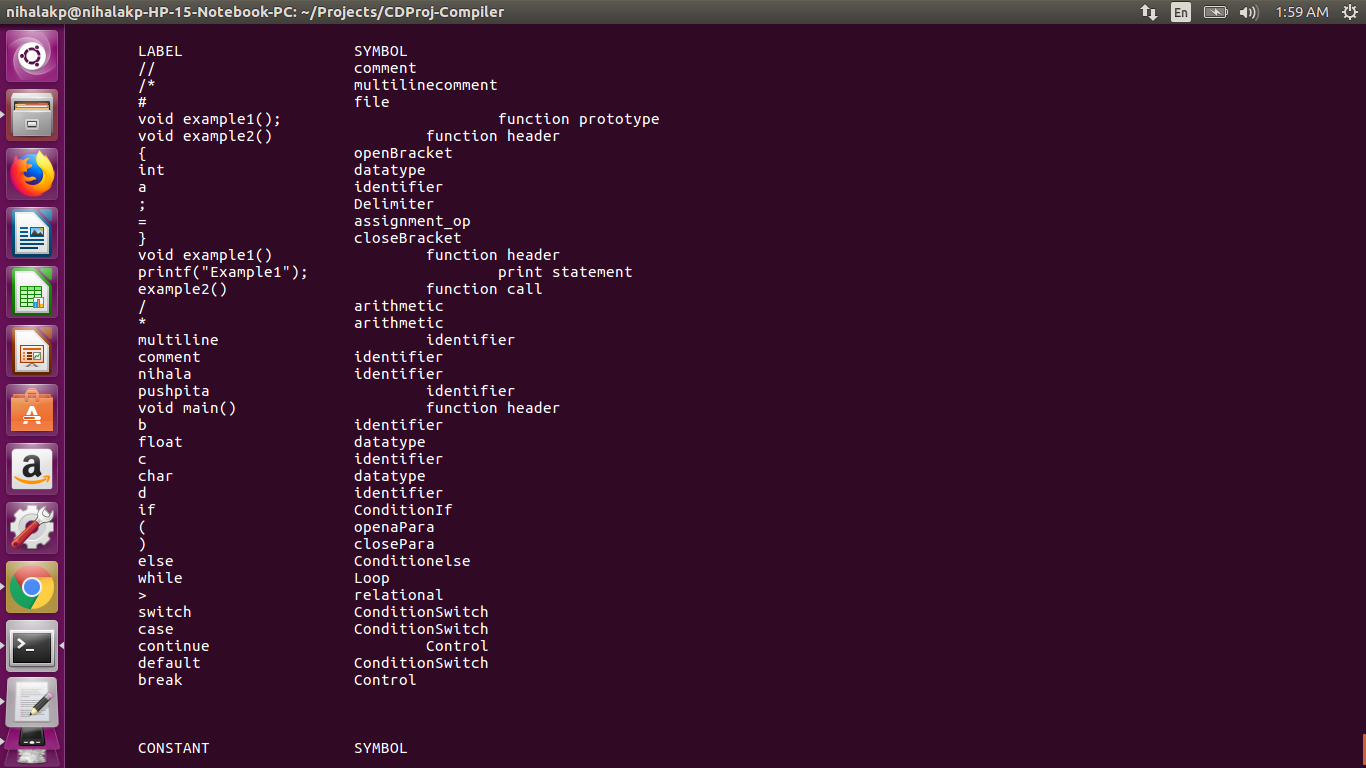
case 1: a = 1 ; continue;

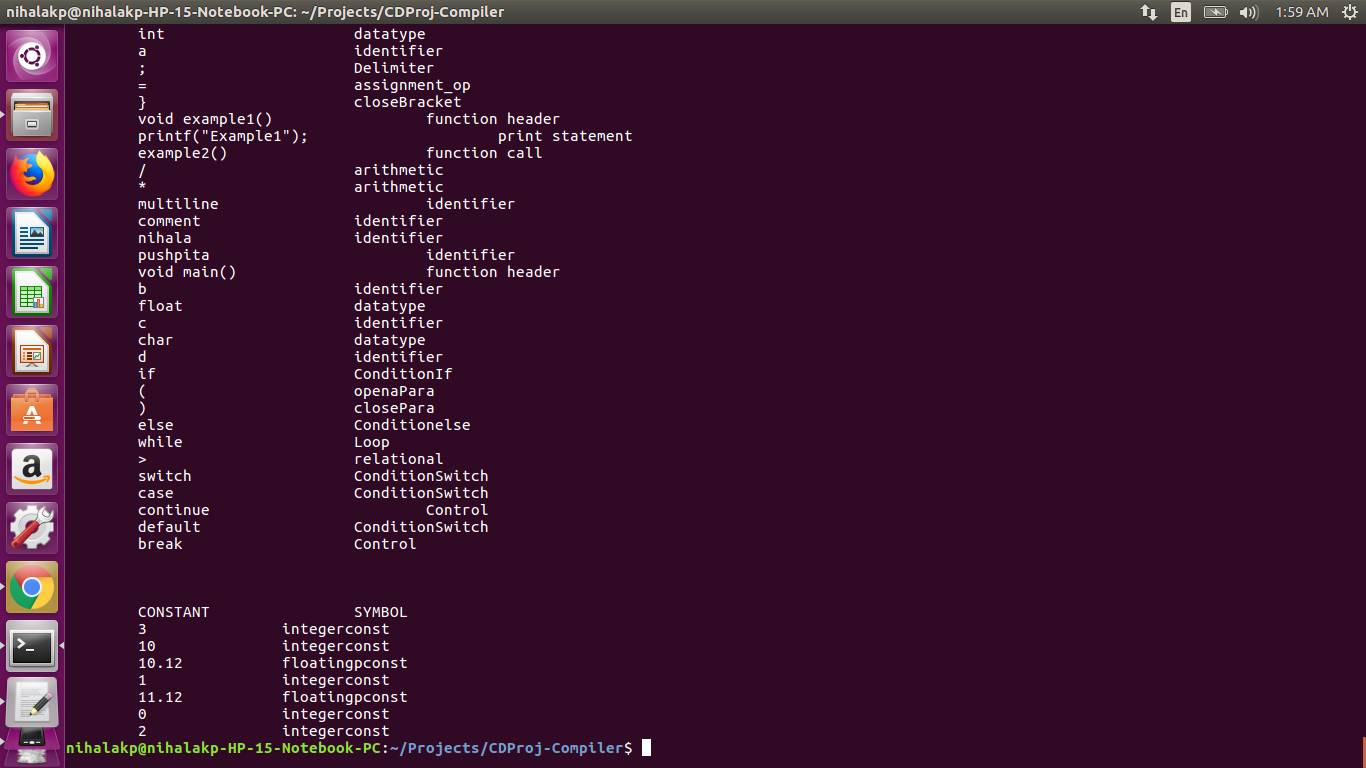
case 2: b = b \* 2 ;

default: break;

}

}





* Every possible token that our lexical analyzer can identify is noted properly, and the constant table is made properly.

# Our Mini-compiler Code

%{

#include "helper.h"

int cnto = 0;

int cntc = 0;

%}

datatype ("int"|"float"|"char"|"struct"|"enum")

intconst [0-9]+

fpconst [+-]?([0-9]\*[.])?[0-9]+

a\_operators [/\*+%]

r\_operators (">"|"<"|">="|"<="|"==")

assignment\_op [=]

identifier [\_a-zA-Z][\_a-zA-Z0-9]{0,30}

ConditionIf ("if")

Conditionelse ("else")

ConditionSwitch("switch"|"case"|"default")

Control ("continue"|"break")

Loop ("for"|"while")

file ^"#include".+$

Comment "//".+$

oBracket \{

oPara \(

cBracket \}

cPara \)

funccall [\_a-zA-Z][\_a-zA-Z0-9]{0,30}\(\)

funcprot ("void"|"int"|"float"|"char")(" "){identifier}\(.\*\)("\;")

funcheader ("void"|"int"|"float"|"char")(" "){identifier}\(.\*\)

multicomment "/\*".+"\*/"

Delimiter "\;"

String ("\"").+("\"")

printf ("printf\("){String}("\)");

%%

{Comment} { insert("//", "comment"); }

{multicomment} {insert("/\*", "multilinecomment");}

{printf} {insert(yytext , "print statement");}

{String} { insert(yytext, " String");}

{file} { insert("#", "file"); }

{funcprot} {insert(yytext, "function prototype");}

{funcheader} {insert(yytext, "function header");}

{funccall} {insert(yytext, "function call");}

{datatype} { insert(yytext, "datatype"); }

{ConditionIf} { insert(yytext, "ConditionIf"); }

{Conditionelse} { insert(yytext, "Conditionelse"); }

{ConditionSwitch} { insert(yytext, "ConditionSwitch"); }

{Control} { insert(yytext, "Control"); }

{Loop} { insert(yytext, "Loop"); }

{Delimiter} { insert(yytext, "Delimiter"); }

{intconst} { insertC(yytext,"integerconst"); }

{fpconst} { insertC(yytext, "floatingpconst"); }

{identifier} { insert(yytext, "identifier"); }

{a\_operators} { insert(yytext, "arithmetic"); }

{r\_operators} { insert(yytext, "relational"); }

{oBracket} { insert(yytext, "openBracket"); ++cnto; }

{cBracket} { insert(yytext, "closeBracket"); ++cntc; }

{oPara} { insert(yytext, "openaPara"); }

{cPara} { insert(yytext, "closePara"); }

{assignment\_op} { insert(yytext, "assignment\_op"); }

[\r] ;

[\n] ;

[\t] ;

: ;

, ;

[ ] ;

. { printf("%s error\n", yytext);}

%%

void main()

{

yyin=fopen("NoErrorFile.txt","r");

yylex();

if(cnto!=cntc)

{

printf("\nParenthesis error!");

}

Display();

printf("\n\n");

DisplayC();

}

int yywrap()

{

return(1);

}